

## Re: [PATCH 8/12] generic hweight{32,16,8}()

---

*Source:* <http://linux.derkeiler.com/Mailing-Lists/Kernel/2006-01/msg11467.html>

---

- *From:* [linux@xxxxxxxxxxx](mailto:linux@xxxxxxxxxxx)
  - *Date:* 31 Jan 2006 11:49:49 -0500
- 

This is an extremely well-known technique. You can see a similar version that uses a multiply for the last few steps at

<http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel>

which refers to

"Software Optimization Guide for AMD Athlon 64 and Opteron Processors"

[http://www.amd.com/us-en/assets/content\\_type/white\\_papers\\_and\\_tech\\_docs/25112.PDF](http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF)

It's section 8.6, "Efficient Implementation of Population-Count Function in 32-bit Mode", pages 179-180.

It uses the name that I am more familiar with, "popcunt" (population count), although "Hamming weight" also makes sense.

Anyway, the proof of correctness proceeds as follows:

```
b = a - ((a >> 1) & 0x55555555);
c = (b & 0x33333333) + ((b >> 2) & 0x33333333);
d = (c + (c >> 4)) & 0x0f0f0f0f;
#if SLOW_MULTIPLY
e = d + (d >> 8)
f = e + (e >> 16);
return f & 63;
#else
/* Useful if multiply takes at most 4 cycles */
return (d * 0x01010101) >> 24;
#endif
```

The input value  $a$  can be thought of as 32 1-bit fields each holding their own hamming weight. Now look at it as 16 2-bit fields.

Each 2-bit field  $a_1..a_0$  has the value  $2*a_1 + a_0$ . This can be converted into the hamming weight of the 2-bit field  $a_1+a_0$  by subtracting  $a_1$ .

That's what the  $(a >> 1) \&$  mask subtraction does. Since there can be no borrows, you can just do it all at once.

Enumerating the 4 possible cases:

```
0b00 = 0 -> 0 - 0 = 0
0b01 = 1 -> 1 - 0 = 1
```

Re: [PATCH 8/12] generic hweight{32,16,8}()

0b10 = 2 -> 2 - 1 = 1  
0b11 = 3 -> 3 - 1 = 2

The next step consists of breaking up b (made of 16 2-bit fields) into even and odd halves and adding them into 4-bit fields. Since the largest possible sum is 2+2 = 4, which will not fit into a 4-bit field, the 2-bit fields have to be masked before they are added.

After this point, the masking can be delayed. Each 4-bit field holds a population count from 0..4, taking at most 3 bits. These numbers can be added without overflowing a 4-bit field, so we can compute  $c + (c \gg 4)$ , and only then mask off the unwanted bits.

This produces d, a number of 4 8-bit fields, each in the range 0..8.  
>From this point, we can shift and add d multiple times without overflowing an 8-bit field, and only do a final mask at the end.

The number to mask with has to be at least 63 (so that 32 on't be truncated), but can also be 128 or 255. The x86 has a special encoding for signed immediate byte values -128..127, so the value of 255 is slower. On other processors, a special "sign extend byte" instruction might be faster.

On a processor with fast integer multiplies (Athlon but not P4), you can reduce the final few serially dependent instructions to a single integer multiply. Consider d to be 3 8-bit values d3, d2, d1 and d0, each in the range 0..8. The multiply forms the partial products:

```
d3 d2 d1 d0
d3 d2 d1 d0
d3 d2 d1 d0
+ d3 d2 d1 d0
-----
e3 e2 e1 e0
```

Where  $e3 = d3 + d2 + d1 + d0$ . e2, e1 and e0 obviously cannot generate any carries.

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@xxxxxxxxxxxxxxxxxxx  
More majordomo info at <http://vger.kernel.org/majordomo-info.html>  
Please read the FAQ at <http://www.tux.org/lkml/>

- 
- *Follow-Ups:*
    - ◆ **Re: [PATCH 8/12] generic hweight{32,16,8}()**
    - ◇ *From:* Grant Grundler

Re: [PATCH 8/12] generic hweight{32,16,8}()

- Prev by Date: [\*Re: CD writing in future Linux try #2\*](#)
- Next by Date: [\*Re: \[PATCH 0/11\] LED Class, Triggers and Drivers\*](#)
- Previous by thread: [\*\[2.6 patch\] wrong firmware location in IPW2100 Kconfig entry\*](#)
- Next by thread: [\*Re: \[PATCH 8/12\] generic hweight{32,16,8}\(\)\*](#)
- Index(es):
  - ◆ [\*Date\*](#)
  - ◆ [\*Thread\*](#)